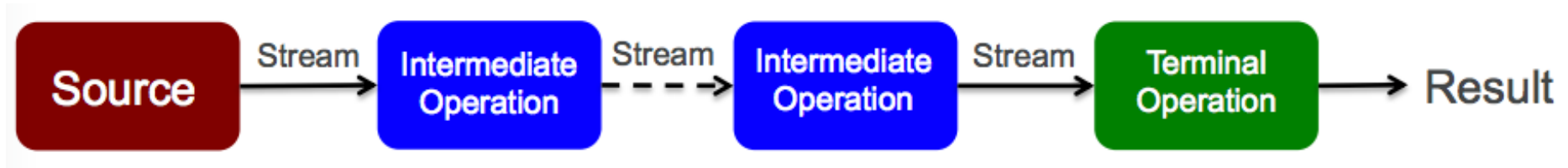


Stream API



Índice

Introducción a los conceptos de programación funcional

Elementos de un Stream

Tipos primitivos de objetos Stream

Fuentes de Stream en JDK 8

Interface Stream: Operaciones intermedias

Interface Stream: Operaciones Terminales

Clase Optional

Programación imperativa

Nombres y valores

Uso de variables como asociación entre nombres y valores

Uso de secuencias de comandos

- Cada comando consiste de una asignación
- Puede modificar el valor de variables
- Form `<nombre_variable> = <expresion>`
- Las expresiones pueden referirse a otras variables
 - Cuyos valores pueden haber sido cambiados por comandos precedentes
- Los valores pueden por lo tanto, ser pasados de un comando a otro
- Los comandos pueden ser repetidos en los bucles

Programación funcional

Nombres y valores

Basado en llamados a funciones estructuradas

El llamado a una función llama otras funciones (composición)

`<function1>(<function2>(<function3> ...> ...)`

Cada una de las funciones recibe valores de la función que la invoca y a su vez proporciona valores de regreso a dicha función

Los nombres son utilizados como parámetros formales

- Una vez que un valor es asignado este no puede ser modificado

no hay concepto de comando, como en el código imperativo

- Por lo tanto no existe el concepto de repetición

Nombres y Valores

Imperativo

- El mismo nombre puede ser asociado con diferentes valores

Funcional

- Un nombre es asociado con un valor

Orden de ejecución

Imperativo

- Los valores asociados con nombres pueden ser modificados
- El orden de ejecución de comandos establece un contrato
 - Si es modificado, el comportamiento de la aplicación podría cambiar

Funcional

- Los valores asociados con nombres no pueden ser cambiados
- El orden de ejecución no tiene impacto en el resultado
- No existe un orden de ejecución preestablecido

Repetición

Imperativo

- los valores asociados con nombres pueden ser modificados por comandos
- los comandos pueden ser repetidos conduciéndonos a cambios repetidos
- nuevos valores pueden ser asociados con el mismo nombre a través de repetición (bucles)

Funcional

- los valores asociados con nombres no deben ser modificados
- Cambios repetitivos son logrados anidando llamados a funciones
- los nuevos valores pueden ser asociados al mismo nombre a través de la recursión

Funciones como valores

La programación funcional, permite que las funciones sean tratadas como valores

- Esta es la razón por la que se necesitan las expresiones lambda
- Para lograr que la programación sea mas sencilla que las clases internas anónimas

Conclusiones

La programación imperativa y funcional son diferentes

Imperativa

- Los valores asociados con nombres pueden ser modificados
- El orden de ejecución es definido en un contrato
- la repetición es explícita y externa

Funcional

- Los valores asociados con nombres son establecidos una vez y no pueden ser modificados
- El orden de ejecución no está definido
- La repetición se realiza mediante el uso de la recursión

Elementos de un Stream

Stream Overview

Abstracción para especificar cálculos agregados

- No es una estructura de datos
- Puede ser infinita

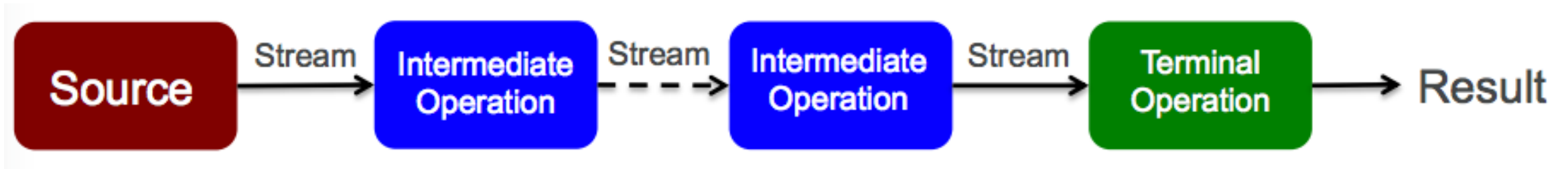
Simplifica la descripción de cálculos agregados

- Expone oportunidades de optimización
- laziness y paralelismo

Descripción de Stream

Una tubería de 'stream' consiste de tres tipos de cosas:

- Una fuente
- Cero o más operaciones intermedias
- Una operación terminal
 - Genera un resultado o un efecto secundario

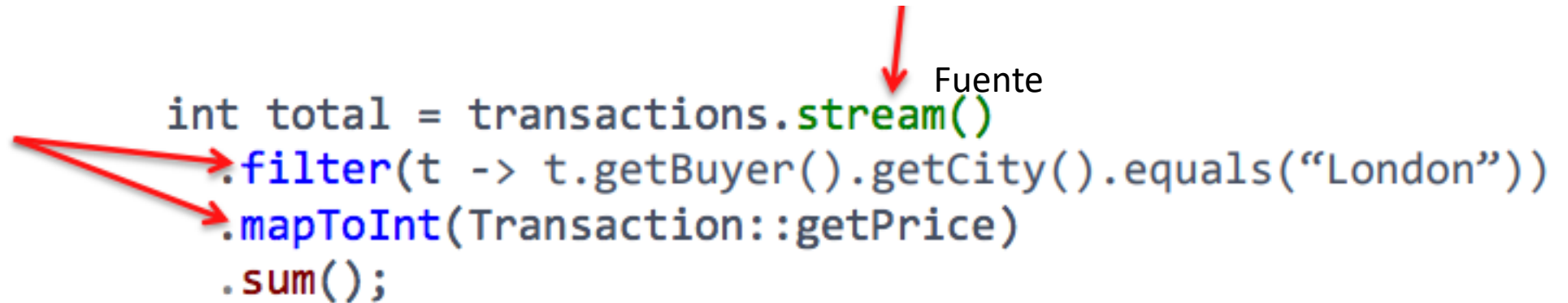


Descripción de Stream

Ejemplo:

Operaciones
intermedias

```
int total = transactions.stream()  
                .filter(t -> t.getBuyer().getCity().equals("London"))  
                .mapToInt(Transaction::getPrice)  
                .sum();
```



Fuente

Operación terminal

Operaciones terminales

La tubería es evaluada cuando se invoca la operación terminal

- Todas las operaciones pueden ser ejecutadas secuencialmente o en paralelo
- las operaciones intermedias pueden ser unidas
 - Evitando pases redundantes sobre los datos
 - Operaciones de corto-circuito (ejemplo findFirst)
 - Evaluación perezosa (Lazy)

Conclusiones

Considere al Stream como una tubería

Procesamiento de datos proporcionados por la fuente

- No hay uso explícito de bucles
- lo cual implica que un Stream puede ser creado en paralelo

Streams de Objetos y tipos primitivos

Objetos y primitivas

Resumen

El lenguaje Java no es realmente orientado a objetos

Se incluyen los tipos primitivos

- byte, short, int, long, double, float char

En algunas situaciones se encapsulan como objetos

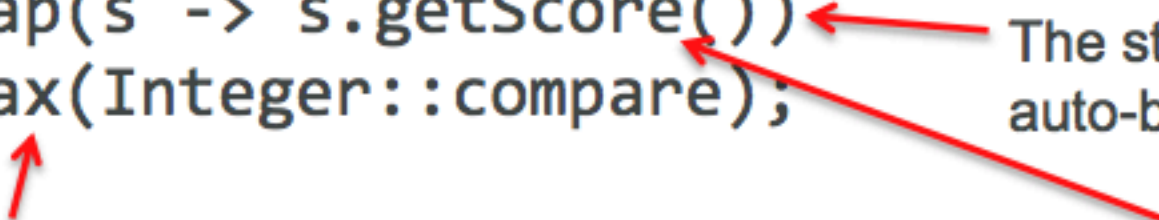
- Por ejemplo en colecciones
- Byte, Short, Integer, ...

La conversión entre tipos primitivos y representación de objetos es muy a menudo hecho por el auto-boxing y unboxing

Objetos Stream

- By default, a stream produces elements that are objects
- Sometimes, this is not the best solution

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .map(s -> s.getScore())  
    .max(Integer::compare);
```



max() debe desempacar
(unbox) cada objeto Integer
para obtener el valor

getScore() devuelve un tipo primitivo (int)


Streams Primitivos

Para evitar la creación innecesaria de objetos, se cuenta con tres tipos primitivos de stream:

- IntStream, DoubleStream y LongStream

Estos pueden ser utilizados con ciertas operaciones

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .mapToInt(s -> s.getScore())  
    .max();
```



El stream que devuelve mapToInt es un stream de valores tipo int,.
No se requiere boxing and unboxing

Conclusiones

El lenguaje Java tiene valores de tipos primitivos así como tipos de objetos

Para mejorar la eficiencia de stream tenemos tres tipos de primitivas stream

- IntStream, DoubleStream y LongStream

Utilice métodos tales como `mapToInt()`, `mapToDouble()` y `mapToLong()`

Fuentes de Stream en JDK 8

Bibliotecas JDK 8

Existen 95 métodos en 23 clases que regresan un Stream

71 métodos en 15 clases pueden ser utilizadas como fuentes Stream prácticas

Interface Collection

`stream()`

- Proporciona un stream secuencial de elementos en la colección

`parallelStream()`

- Proporciona un Stream paralelo de elementos en la colección
- Utiliza el marco de trabajo fork-join

Clases de Array

stream()

- Un array es una colección de datos. Por lo que es capaz de crear un stream
- Proporciona un stream secuencial
- métodos sobrecargado para diferentes tipos
 - double, int, long, Object

Clases File

`find(Path, BiPredicate, FileVisitOption)`

- Un Stream de tipo File referencia que coinciden un BiPredicado dado

`list(Path)`

- Un Stream de entradas de un directorio dado

`lines(Path)`

- Un Stream de strings que son las líneas leídas de un fichero dado

`walk(Path, FileVisitOption)`

- Un stream de referencias de tipo File

Números aleatorios

Tres clases relacionadas

- Random, ThreadLocalRandom, SplittableRandom

Métodos para producir streams finitos o infinitos de números aleatorios

- ints(), doubles(), longs()
- cuatro versiones de cada una
 - Finito o infinito
 - Con y sin semilla

Clases misceláneas y métodos

JarFile/ZipFile: stream ()

- Regresa un stream de tipo File con el contenido de los ficheros comprimidos

BufferedReader: Lines()

- Regresa unStream de strings que son las. líneas leídas de la entrada

Pattern; splitAsStream()

- Regresa un stream de strings de coincidencias de un patrón
- Al igual que split(), pero regresa un stream en vez de un array

Clases misceláneas y métodos

CharSequence

- `chars()`: Char regresa una secuencia de valores de tipo `int`
- `codePoints()`: Code apunta a valores de esta secuencia

BitSet

- `stream()`: indices de bits que son establecidos

Métodos estáticos

IntStreams, DoubleStream, LongStream

Estas interfaces son primitivas especializadas de la interfaz Stream

`concat(Stream, Stream), empty()`

- Concatena dos streams especializadas, regresa un stream vacío

`of (T... values)`

- Un stream que consiste en los valores especificados

`range(int, int), rangeClosed(int, int)`

- Un stream desde inicio hasta fin (exclusivo e inclusivo)

`generate(IntSupplier), iterate(int, IntUnaryOperator)`

- Un stream infinito creado por un proveedor dado
- `iterate()` utiliza una semilla para inicializar el stream

Conclusiones

Muchos lugares para obtener stream que son fuentes

- Métodos útiles para recuperar líneas procedentes de ficheros, ficheros de archivos, etc.

Solo la Collection puede proporcionar un stream paralelo de forma directa

Stream Interface: Intermediate Operations

Stream Interface

Un Stream proporciona una secuencia de elementos

- Soportan operaciones de agregado secuenciales o en paralelo

La mayoría de la operaciones requieren un parámetro que describe su comportamiento

- En general utilizando expresiones lambda
- La mayoría no modifican el stream (Non-Interfering)
- En gneral son sin estado

Los streams pueden ser modificados de secuencial a paralelo (y viceversa)

- Todo el procesamiento es realizado ya sea secuencialmente o en paralelo

Filtrado y Mapeo

`distinct()`

- Regresa un flujo sin elementos duplicados

`filter(Predicado p)`

- Regresa un flujo con aquellos elementos que cumplen el predicado

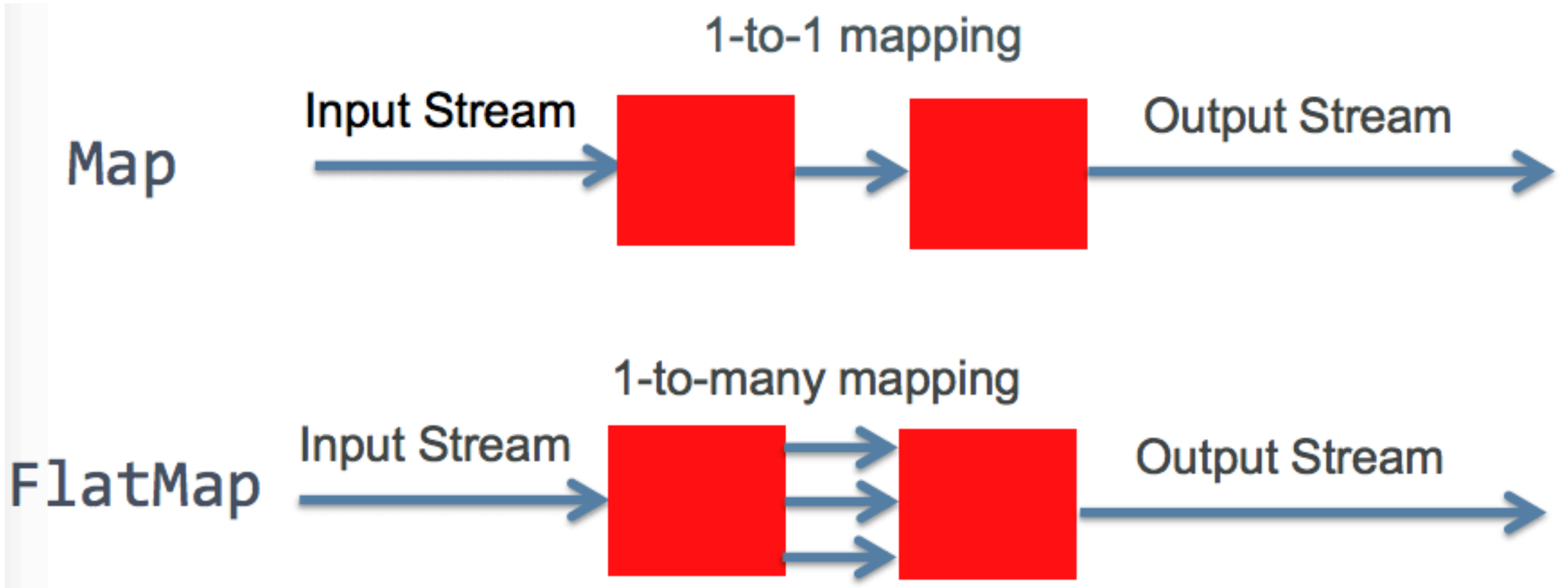
`map(Function p)`

- Regresa un stream en donde la función proporcionada es aplicada a cada uno de los elementos del stream

`mapToInt(), mapToDouble(), mapToLong()`

- Funcionan como `map()` pero producen streams de primitivas en vez de objetos

Maps and FlatMaps



Ejemplo de FlatMap

Palabras en un fichero

```
List<String> output = reader
    .lines()
    .flatMap(line -> Stream.of(line.split(REGEXP)))
    .filter(word -> word.length() > 0)
    .collect(Collectors.toList());
```

Restringiendo el tamaño de un Stream

`skip(long n)`

- Regresa un stream que excluye los primeros n elementos del stream de entrada

`limit(long n)`

- Regresa un stream que unicamente contiene los primeros n elementos del stream de entrada

```
String output = bufferedReader  
    .lines()  
    .skip(2)  
    .limit(2)  
    .collect(Collectors.joining());
```

Ordenación y desordenación

A medida que van pasando

`sorted(Comparator c)`

- Regresa un stream que es ordenado, el comparador determina la ordenación
- `sorted()` sin argumentos ordena en base a una ordenación natural

`unordered()`

- Heredado de `BaseStream`
- Regresa un stream que está desordenado
- Puede mejorar la eficiencia de operaciones tales como `distinct()` y `groupingBy()`

Orservación de elementos de Stream

`peek(Consumer c)`

- Regresa un stream de salida que es idéntico al input de entrada
- Cada uno de los elemento es pasado por el método `accept()` de `Consumer`
- El `Consumer` no debe modificar los elementos del stream
- Es útil para depuración

Conclusiones

La interfaz Stream representa operaciones de agregado sobre elementos

La mayoría de los métodos pueden utilizar expresiones Lambda para definir comportamiento

Poderosas operaciones intermedias que permite manipular los streams

- Se pueden construir procesamientos complejos a partir de bloque de construcción simples

Interfaz Stream: Operaciones terminales

Operaciones Terminales

Finaliza la tubería de operaciones sobre el stream

Únicamente hasta este punto se realizan los procesamientos

- Esto permite la optimización de la tubería
- Evaluación perezosa (Lazy)
- Operaciones de fusionado o de fusionado
- Eliminación de operaciones redundantes
- Ejecución en paralelo

Genera un resultado explícito o un efecto secundario

Elementos coincidentes

findFirst()

- La primer coincidencia

findAny()

- Trabaja igual que findFirst(), pero actúa sobre un stream paralelo
- boolean allMatch(Predicate p)
 - Si todos los elementos del stream coinciden utilizando el Predicado p
- boolean anyMatch(Predicate p)
 - Si al menos uno de los elementos coincide con la condición del predicado p
- boolean noneMatch(Predicate p)
 - ninguno de los elementos coincide de acuerdo al predicado

Recolección de resultados

A medida que van pasando

`collect(Collector c)`

- Realiza una reducción mutable sobre el stream

`toArray()`

- regresa un array que contiene los elementos del stream

Resultados Numéricos

A medida que van pasando

`count()`

- Regresa el numero de elementos que hay en el stream

`max (Comparator c)`

- El elemento con el máximo valor que está dentro del stream en base al comparador

`min(Comparador c)`

- El elemento con el valor más pequeño que está dentro del stream en base al comparador
- Regresa un Optional, si el stream esta vacío

Resultados Numéricos

Streams de tipo primitivo (IntStream, DoubleStream(), LongStream)

average()

- Regresa la media aritmética del stream
- Si el stream está vacío. regresa un Optional

sum()

- Regresa la suma de los elementos del stream

Iteración

`forEach(Consumer c)`

- Realiza una acción por cada elemento del stream

`forEachOrdered(Consumer c)`

- Funciona igual que `forEach`, pero asegura que el orden de los elementos (Si hay elementos) es respetado cuando se utiliza por un stream paralelo

Utilízelo con precaución

- Alienta el estilo de programación no-funcional (imperativo)

Plegando una secuencia

Creción de un resultado a partir de varios elementos de entrada

`reduce(BinaryOperator accumulator)`

- realiza una reducción utilizando el Operador Binario
- El acumulador toma un resultado parcial y el elemento next, y regresa un nuevo resultado parcial
- Regresa un Optional
- Dos versiones
 - Una que toma un valor inicial (no regresa Optional)
 - Una que toma un valor inicial y una BiFuncion (equivalente a un fused map and reduce)

Conclusiones

Las operaciones de tipo Terminal proporcionan resultados o efectos secundarios

Se dispone de muchos tipos de operación

Los del tipo reduce and collect necesitan ser observados con más detalle

La clase Optional

Problemas de null

Ciertas situaciones en Java regresan como resultado un valor null

- El cual referencia a un objeto que no ha sido inicializado

Evitar NullPointerException

```
String direction = gpsData.getPosition().getLatitude().getDirection();
```

```
String direction = "UNKNOWN";
```

```
if (gpsData != null) {  
    Position p = gpsData.getPosition();  
  
    if (p != null) {  
        Latitude latitude = p.getLatitude();  
  
        if (latitude != null)  
            direction = latitude.getDirection();  
    }  
}
```

Clase Optional

Ayuda a eliminar la excepcion `NullPointerException`

Las operaciones terminales como `min()` , `max()` podrían no poder regresar un resultado directo

- Suponga que el stream de entrada esta vacío

`Optional <T>`

- Contenedor para una referencia de objeto (null o objeto real)
- Considerelo como un stream con 0 o 1 elemento
- Garantizado que la referencia `Optional` regresada no es null

Optional ifPresent()

```
if (x != null) {  
    print(x);  
}
```

```
opt.ifPresent(x -> print(x));  
opt.ifPresent(this::print);
```

Optional filter()

rechaza ciertos valores del Optional

```
if (x != null && x.contains("a")) {  
    print(x);  
}
```

```
opt.filter(x -> x.contains("a"))  
    .ifPresent(this::print);
```

Optional map()

Transforma un valor si lo hay

```
if (x != null) {  
    String t = x.trim();  
    if (t.length() > 0)  
        print(t);  
}
```

```
opt.map(String::trim)  
    .filter(t -> t.length() > 0)  
    .ifPresent(this::print);
```

Optional flatMap()

En profundidad

```
public String findSimilar(String s)
```

```
Optional<String> tryFindSimilar(String s)
```

```
Optional<Optional<String>> bad = opt.map(this::tryFindSimilar);  
Optional<String> similar = opt.flatMap(this::tryFindSimilar);
```


Conclusiones

La clase Optional elimina los problemas de NullPointerException

Puede ser utilizado de diferentes formas para proporcionar el manejo condicional complejo

Conclusiones de la Introducción a Stream

Los Streams proporcionan una forma directa para el estilo de programación funcional en Java

Los streams pueden ser objetos o objetos de tipo primitivo

Un stream consiste de una fuente, posibles operaciones intermedias y una operación terminal

- Ciertas operaciones de tipo Terminal regresan un Objeto Optional para evitar posibles problemas generados por NullPointerExceptions